Set-theoretic Foundation of Parametric Polymorphism and Subtyping

Giuseppe Castagna\textsuperscript{1} and Zhiwu Xu\textsuperscript{1,2}

\textsuperscript{1}CNRS, Laboratoire Preuves, Programmes et Systèmes, Université Paris Diderot, Paris, France.

\textsuperscript{2}State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing, China

ICFP, Tokyo, 19th of September, 2011
Goal

1. Take your favorite **type constructors**
   \[
   \times, \to, \{\ldots\}, \text{chan()}, \ldots
   \]
Goal

1. Take your favorite **type constructors**
   - $\times$, $\to$, $\{\ldots\}$, $\text{chan}()$, ...

2. Add **Boolean connectives**:
   - $\lor$, $\land$, $\neg$
Goal

1. Take your favorite type constructors
   \(\times, \rightarrow, \{\ldots\}, \text{chan}(), \ldots\)

2. add Boolean connectives:
   \(\lor, \land, \neg\)

3. add type variables
   \(\alpha, \beta, \gamma, \ldots\)
Goal

1. Take your favorite type constructors
   \( \times, \rightarrow, \{\ldots\}, \text{chan}(), \ldots \)

2. add Boolean connectives:
   \( \lor, \land, \neg \)

3. add type variables
   \( \alpha, \beta, \gamma, \ldots \)

4. give an intuitive (ie, set-theoretic) semantics so as to deduce
   - classic distribution laws (for all \( \alpha, \beta, \gamma \))
     \[(\alpha \lor \beta) \times \gamma \subseteq (\alpha \times \gamma) \lor (\beta \times \gamma)\]
1. **Goal**

1. Take your favorite **type constructors**
   \[ \times, \rightarrow, \{\ldots\}, \text{chan()}, \ldots \]

2. add **Boolean connectives**:
   \[ \lor, \land, \neg \]

3. add **type variables**
   \[ \alpha, \beta, \gamma, \ldots \]

4. give an intuitive (*ie*, set-theoretic) semantics so as to deduce
   - classic distribution laws (for all \( \alpha, \beta, \gamma \))
     \[ \left( (\alpha \lor \beta) \times \gamma \right) \subseteq (\alpha \times \gamma) \lor (\beta \times \gamma) \]
   - data structure containments (for all \( \alpha \)):
     \[ \mu t.(\alpha \times (\alpha \times t)) \lor \text{nil} \leq \mu t.(\alpha \times t) \lor \text{nil} \]
     - \( \alpha \)-lists of even length
     - \( \alpha \)-lists
Goal

1. Take your favorite type constructors
   \( \times, \rightarrow, \{ \ldots \}, \text{chan()}, \ldots \)

2. add Boolean connectives:
   \( \lor, \land, \neg \)

3. add type variables
   \( \alpha, \beta, \gamma, \ldots \)

4. give an intuitive (ie, set-theoretic) semantics so as to deduce
   - classic distribution laws (for all \( \alpha, \beta, \gamma \))
     \[
     ((\alpha \lor \beta) \times \gamma) \leq (\alpha \times \gamma) \lor (\beta \times \gamma)
     \]
   - data structure containments (for all \( \alpha \)):
     \[
     \mu t.(\alpha \times (\alpha \times t)) \lor \text{nil} \leq \mu t.(\alpha \times t) \lor \text{nil}
     \]
     \( \alpha \)-lists of even length \( \leq \alpha \)-lists

WHY?

Giuseppe Castagna and Zhiwu Xu
Set-theoretic Foundation of Parametric Polymorphism and Subtyping
WHY? briefly:

1. Boolean connectives: Unions, products and recursive types encode regular trees and therefore XML. Intersection and negation permit XML typed programming with overloading and powerful pattern matching.

2. Type variables: Parametric polymorphism already demonstrated its worth in practice. Fulfills new needs specific to XML processing (e.g., SOAP envelopes). Sheds new light on the notion of parametricity.
WHY? briefly:

1. **Boolean connectives:**
   - Unions, products and recursive types encode regular trees and therefore **XML**
   - Intersection and negation permit XML typed programming with **overloading** and powerful **pattern matching**.
WHY? briefly:

1. **Boolean connectives:**
   - Unions, products and recursive types encode regular trees and therefore **XML**
   - Intersection and negation permit XML typed programming with **overloading** and powerful **pattern matching**.

2. **Type variables:**
   - Parametric polymorphism already demonstrated its worth in practice.
   - Fulfills new needs specific to XML processing (*eg*, SOAP envelopes).
   - Sheds new light on the notion of **parametricity**.
Real case example: active pages

To create a \textit{dynamically} generated page in the \textit{Ocsigen} web development systems:

\begin{verbatim}
1. define a function from the query string to Xhtml:
   let page_fun(p: {title: string, ...}) : Xhtml = ...

2. bind page fun to the path $\textit{w/index}$ by:
   register new service(page fun, "w/index")

The (wished) type of register new service is
\[ \forall (X \leq \text{Params}. ((X \rightarrow Xhtml) \times \text{Path}) \rightarrow \text{unit} \]
where Params is a specification of all possible query strings
\end{verbatim}
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   ```plaintext
   let page_fun(p: {title: string, ...}) : Xhtml = ...
   ```
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   
   \[
   \text{let page\_fun}(p: \{\text{title: string, ...}\}) : \text{Xhtml} = \ldots
   \]

2. bind `page_fun` to the path `$WEBROOT/w/index` by:

   \[
   \text{register\_new\_service}(\text{page\_fun,"w/index")}
   \]
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to Xhtml:
   
   ```
   let page_fun(p: {title: string, ...}) : Xhtml = ...
   ```

2. bind `page_fun` to the path `$WEBROOT/w/index` by:
   
   ```
   register_new_service(page_fun,"w/index")
   ```

The (wished) type of `register_new_service` is

\[
\forall (X \leq \text{Params}).((X \rightarrow \text{Xhtml}) \times \text{Path}) \rightarrow \text{unit}
\]

where \text{Params} is a specification of all possible query strings
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to *Xhtml*:
   
   \[
   \text{let page\_fun}(p: \{\text{title: string, ...}\}) : \text{Xhtml} = \ldots
   \]

2. bind \text{page\_fun} to the path \$WEBROOT/w/index by:

   \[
   \text{register\_new\_service}(\text{page\_fun},"w/index")
   \]

The (wished) type of \text{register\_new\_service} is

\[
\forall(X \leq \text{Params}).((X \rightarrow \text{Xhtml}) \times \text{Path}) \rightarrow \text{unit}
\]

where \text{Params} is a specification of all possible query strings.
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to *Xhtml*:
   
   ```
   let page_fun(p: {title: string, ...}) : Xhtml = ...
   ```

2. bind `page_fun` to the path `$WEBROOT/w/index` by:
   
   ```
   register_new_service(page_fun,"w/index")
   ```

The (wished) type of `register_new_service` is

\[ \forall (X \leq \text{Params}).((X \rightarrow Xhtml) \times \text{Path}) \rightarrow \text{unit} \]

where `Params` is a specification of all possible query strings.
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to *Xhtml*:
   
   ```
   let page_fun(p: {title: string, ...}) : Xhtml = ...
   ```

2. bind `page_fun` to the path `$WEBROOT/w/index` by:
   
   ```
   register_new_service(page_fun, "w/index")
   ```

The (wished) type of `register_new_service` is

\[ \forall (X \leq \text{Params}).((X \to \text{Xhtml}) \times \text{Path}) \to \text{unit} \]

where `Params` is a specification of all possible query strings.
Real case example: active pages

To create a *dynamically* generated page in the *Ocsigen* web development systems:

1. define a function from the *query string* to *Xhtml*:
   ```
   let page_fun(p: {title: string, ...}) : Xhtml = ...
   ```

2. bind `page_fun` to the path `$WEBROOT/w/index` by:

   ```
   register_new_service(page_fun,"w/index")
   ```

The (wished) type of `register_new_service` is

$$\forall (X \leq \text{Params}).((X \rightarrow \text{Xhtml}) \times \text{Path}) \rightarrow \text{unit}$$

*type variables*  

*XML types*  

*bounded quantification*  

*higher-order functions*
Real case example: active pages

To create a dynamically generated page in the Ocsigen web development systems:

1. define a function from the query string to Xhtml:
   
   \[
   \text{let \ page\_fun(p: \{title: \text{string}, \ldots\}) : \text{Xhtml} = \ldots}
   \]

2. bind \text{page\_fun} to the path $\text{\$WEBROOT/w/index}$ by:
   
   \[
   \text{register\_new\_service(page\_fun, "w/index")}
   \]

Since all these features are not available, Ocsigen's type system must be unplugged.

The type of \text{register\_new\_service} is

\[
\forall (X \leq \text{Params}).((X \rightarrow \text{Xhtml}) \times \text{Path}) \rightarrow \text{unit}
\]

**Type variables**

**XML types**

**Bounded quantification**

**Higher-order functions**
Current status

Study of a type system of (recursive/regular) types with

\[
t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha
\]

- **type constructors**
- **logical connectives**
- **type variables**
Current status

Study of a type system of (recursive/regular) types with

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

- **Logical connectives**: Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation.*
Current status

Study of a type system of (recursive/regular) types with

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

type constructors  logical connectives  type variables

- **Logical connectives**: Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

  **Semantic subtyping**
  (implemented by the language CDuce).
Current status

Study of a type system of (recursive/regular) types with

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

- **Logical connectives:** Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

  Semantic subtyping
  (implemented by the language CDuce).

- **Type variables:** A set-theoretic approach was deemed unfeasible or even impossible:
Current status

Study of a type system of (recursive/regular) types with

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

- **Logical connectives:** Well-known how to implement a functional language with pattern-matching, higher-order functions, and *connectives with set theoretic interpretation*.

**Semantic subtyping**
(implemented by the language CDuce).

- **Type variables:** A set-theoretic approach was deemed unfeasible or even impossible:

**This work**
(built on the work of semantic subtyping)
Semantic Subtyping in a nutshell
Semantic subtyping

\[ t ::= B | t \times t | t \rightarrow t | t \lor t | t \land t | \neg t | 0 | 1 \]
Semantic subtyping

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

- **Constructor subtyping is easy:**
  constructors do not mix, eg.:

\[
\frac{s_2 \leq s_1 \quad t_1 \leq t_2}{s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2}
\]
Semantic subtyping

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

- **Constructor subtyping** is *easy*:
  constructors do not mix, e.g.:

\[
\begin{align*}
s_2 \leq s_1 & \quad t_1 \leq t_2 \\
\frac{}{s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2}
\end{align*}
\]

- **Connective subtyping** is *harder*:
  connectives distribute over constructors, e.g.:

\[
(s_1 \lor s_2) \rightarrow t \quad \supseteq \quad (s_1 \rightarrow t) \land (s_2 \rightarrow t)
\]
Semantic subtyping

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]

- **Constructor subtyping** is *easy*:
  constructors do not mix, *eg*:

  \[
  s_2 \leq s_1 \quad t_1 \leq t_2 \quad \quad \Rightarrow \quad \quad s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2
  \]

- **Connective subtyping** is *harder*:
  *connectives* distribute over *constructors*, *eg*:

  \[
  (s_1 \lor s_2) \rightarrow t \quad \Rightarrow \quad (s_1 \rightarrow t) \land (s_2 \rightarrow t)
  \]

Define subtyping semantically: [Hosoya, Pierce]

1. Interpret types as sets (of values)
2. **Define** subtyping as set containment.
First, define an interpretation of types into sets.

$$\llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})$$

such that
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ \llbracket \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  \begin{align*}
  \llbracket 0 \rrbracket &= \emptyset \\
  \llbracket t_1 \lor t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
  \llbracket t_1 \land t_2 \rrbracket &= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
  \end{align*}
  \]
First, define an interpretation of types into sets.

\[
\boxed{\text{[ ] : Types } \rightarrow \mathcal{P}(\mathcal{D})}
\]

such that

- **Connectives** have their set-theoretic interpretation:
  
  \[
  \begin{align*}
  \text{[}\emptyset\text{]} &= \emptyset \\
  \text{[}t_1 \lor t_2\text{]} &= \text{[}t_1\text{]} \cup \text{[}t_2\text{]} \\
  \text{[}\neg t\text{]} &= \mathcal{D} \setminus \text{[}t\text{]} \\
  \text{[}t_1 \land t_2\text{]} &= \text{[}t_1\text{]} \cap \text{[}t_2\text{]}
  \end{align*}
  \]

- **Constructors** have their natural interpretation:

  \[
  \begin{align*}
  \text{[}t_1 \times t_2\text{]} &= \text{[}t_1\text{]} \times \text{[}t_2\text{]} \\
  \text{[}t_1 \rightarrow t_2\text{]} &= \{ f \mid f \text{ function from [}t_1\text{] to [}t_2\text{]} \}
  \end{align*}
  \]
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[
\llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})
\]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  \begin{align*}
  \llbracket 0 \rrbracket &= \emptyset \\
  \llbracket t_1 \lor t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
  \llbracket t_1 \land t_2 \rrbracket &= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
  \end{align*}
  \]

- **Constructors** have their natural interpretation:
  \[
  \begin{align*}
  \llbracket t_1 \times t_2 \rrbracket &= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
  \llbracket t_1 \rightarrow t_2 \rrbracket &= \{ f \mid f \text{ function from} \llbracket t_1 \rrbracket \text{ to} \llbracket t_2 \rrbracket \}
  \end{align*}
  \]

Then define the subtyping relation as set-containment.

\[
s \leq t \overset{\text{def}}{\iff} \llbracket s \rrbracket \subseteq \llbracket t \rrbracket
\]
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [ \ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[ [0] = \emptyset \]
  \[ [t_1 \lor t_2] = [t_1] \cup [t_2] \]
  \[ [\neg t] = \mathcal{D} \setminus [t] \]
  \[ [t_1 \land t_2] = [t_1] \cap [t_2] \]

- **Constructors** have their natural interpretation:
  \[ [t_1 \times t_2] = [t_1] \times [t_2] \]
  \[ [t_1 \rightarrow t_2] = \{ f \mid f \text{ function from } [t_1] \text{ to } [t_2] \} \]

Then *define* the subtyping relation as set-containment.

\[ s \leq t \overset{\text{def}}{\iff} [s] \subseteq [t] \]
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [\ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  
  \[
  [\mathbf{0}] = \emptyset \\
  [\neg t] = \mathcal{D} \setminus [t] \\
  [t_1 \lor t_2] = [t_1] \cup [t_2] \\
  [t_1 \land t_2] = [t_1] \cap [t_2] \\
  \]

- **Constructors** have their natural interpretation:
  
  \[
  [t_1 \times t_2] = [t_1] \times [t_2] \\
  [t_1 \rightarrow t_2] = \{ f \mid f \text{ function from } [t_1] \text{ to } [t_2] \} \\
  \]

Then **define** the subtyping relation as set-containment.

\[ s \leq t \overset{\text{def}}{\iff} [s] \subseteq [t] \]
First, define an interpretation of types into sets.

\[ [\ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D}) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  [0] = \emptyset \\
  [-t] = D \setminus [t] \\
  [t_1 \lor t_2] = [t_1] \cup [t_2] \\
  [\neg t] = D \setminus [t] \\
  [t_1 \land t_2] = [t_1] \cap [t_2]
  \]

- **Constructors** have their natural interpretation:
  \[
  [t_1 \times t_2] = [t_1] \times [t_2] \\
  [t_1 \rightarrow t_2] = \{ f \mid f \text{ function from} [t_1] \text{ to} [t_2] \}
  \]

Then define the subtyping relation as set-containment.

\[ s \leq t \iff [s] \subseteq [t] \]

Key idea:

**Do not define what types are**
**define how they are related**
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[
\llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})
\]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  \llbracket 0 \rrbracket = \emptyset \\
  \llbracket t_1 \lor t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket = \mathcal{D} \setminus \llbracket t \rrbracket \\
  \llbracket t_1 \land t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
\]

- ** Constructors** have their natural interpretation:
  \[
  \llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
  \llbracket t_1 \rightarrow t_2 \rrbracket = \{ f \mid f \text{ function from} \llbracket t_1 \rrbracket \text{ to} \llbracket t_2 \rrbracket \}
\]

Then define the subtyping relation as set-containment.

\[
s \leq t \iff \llbracket s \rrbracket \subseteq \llbracket t \rrbracket
\]

**Key idea**

Do not define what types are
define how they are related
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[
\llbracket \ \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})
\]

such that

Connectives have their set-theoretic interpretation:

\[
\begin{align*}
\llbracket 0 \rrbracket &= \emptyset \\
\llbracket t_1 \lor t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
\llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
\llbracket t_1 \land t_2 \rrbracket &= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
\end{align*}
\]

Constructors have their natural interpretation:

\[
\begin{align*}
\llbracket t_1 \times t_2 \rrbracket &= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
\llbracket t_1 \rightarrow t_2 \rrbracket &= \{ f \subseteq \mathcal{D}^2 \mid (d_1, d_2) \in f, d_1 \in \llbracket t_1 \rrbracket \Rightarrow d_2 \in \llbracket t_2 \rrbracket \}
\end{align*}
\]

Then define the subtyping relation as set-containment.

\[
s \leq t \quad \overset{\text{def}}{\iff} \quad \llbracket s \rrbracket \subseteq \llbracket t \rrbracket
\]

Key idea

Do not define what types are, define how they are related.
Semantic subtyping: formalization

1. **First**, define an interpretation of types into sets.

\[
[\ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})
\]

such that

- **Connectives** have their set-theoretic interpretation:
  - \([0] = \emptyset\)
  - \([t_1 \lor t_2] = [t_1] \cup [t_2]\)
  - \([\neg t] = \mathcal{D} \setminus [t]\)
  - \([t_1 \land t_2] = [t_1] \cap [t_2]\)

- **Constructors** have their natural interpretation:
  - \([t_1 \times t_2] = [t_1] \times [t_2]\)
  - \([t_1 \rightarrow t_2] = \mathcal{P}([t_1] \times [t_2])\)

2. **Then** *define* the subtyping relation as set-containment.

\[
s \leq t \iff [s] \subseteq [t]
\]

**Key idea**

*Do not define what types are, define how they are related*
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ [ \ ] : \text{Types} \rightarrow \mathcal{P}(D) \]

such that

- **Connectives** have their set-theoretic interpretation:
  \[
  [\emptyset] = \emptyset \\
  [t_1 \lor t_2] = [t_1] \cup [t_2] \\
  [\neg t] = D \setminus [t] \\
  [t_1 \land t_2] = [t_1] \cap [t_2]
  \]

- **Constructors** have their natural interpretation:
  \[
  [t_1 \times t_2] = [t_1] \times [t_2] \\
  [t_1 \rightarrow t_2] = \mathcal{P}([t_1] \times [t_2])
  \]

Then define the subtyping relation as set-containment.

\[
 s \leq t \iff [s] \subseteq [t]
\]

**Key idea**

Do not define what types are, define how they are related.
First, define an interpretation of types into sets.

\[
\llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})
\]

such that

- **Connectives** have their set-theoretic interpretation:
  
  \[
  \begin{align*}
  \llbracket \bot \rrbracket &= \emptyset \\
  \llbracket t_1 \lor t_2 \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket &= \mathcal{D} \setminus \llbracket t \rrbracket \\
  \llbracket t_1 \land t_2 \rrbracket &= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
  \end{align*}
  \]

- **Constructors** have the same \( \subseteq \) as their natural interpretation:
  
  \[
  \begin{align*}
  \llbracket t_1 \times t_2 \rrbracket &= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
  \llbracket t_1 \rightarrow t_2 \rrbracket &= \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)
  \end{align*}
  \]

Then define the subtyping relation as set-containment.

\[
s \leq t \iff \llbracket s \rrbracket \subseteq \llbracket t \rrbracket
\]
Semantic subtyping: formalization

- **First**, define an interpretation of types into sets.

\[ [\ ] : \text{Types} \to \mathcal{P}(D) \]

such that

- **Connectives** have their set-theoretic interpretation:

\[ [\emptyset] = \emptyset \quad [t_1 \lor t_2] = [t_1] \cup [t_2] \]
\[ [\neg t] = D \setminus [t] \quad [t_1 \land t_2] = [t_1] \cap [t_2] \]

- **Constructors** have the same \( \subseteq \) as their natural interpretation:

\[ [s_1 \times s_2] \subseteq [t_1 \times t_2] \iff \{s_1 \times s_2\} \subseteq \{t_1 \times t_2\} \]
\[ [s_1 \to s_2] \subseteq [t_1 \to t_2] \iff \mathcal{P}(\{s_1 \times s_2\}) \subseteq \mathcal{P}(\{t_1 \times t_2\}) \]

- Then define the subtyping relation as set-containment.

\[ s \leq t \quad [s] \subseteq [t] \]

**Key idea**

Do not define what types are

define how they are related
Semantic subtyping: formalization

First, define an interpretation of types into sets.

\[ \llbracket \cdot \rrbracket : \text{Types} \rightarrow \mathcal{P}(D) \]

such that

- **Connectives** have their set-theoretic interpretation:

  \[
  \llbracket \emptyset \rrbracket = \emptyset \quad \llbracket t_1 \lor t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\
  \llbracket \neg t \rrbracket = D \setminus \llbracket t \rrbracket \quad \llbracket t_1 \land t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket
  \]

- **Constructors** have the same \( \subseteq \) as their natural interpretation:

  \[
  \llbracket s_1 \times s_2 \rrbracket \subseteq \llbracket t_1 \times t_2 \rrbracket \iff \llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket \subseteq \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \\
  \llbracket s_1 \rightarrow s_2 \rrbracket \subseteq \llbracket t_1 \rightarrow t_2 \rrbracket \iff \mathcal{P}(\llbracket s_1 \rrbracket \times \llbracket s_2 \rrbracket) \subseteq \mathcal{P}(\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket)
  \]

Then define the subtyping relation as set-containment.

\[ s \leq t \iff \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \]

**Semantic subtyping**

[Benzaken, Castagna, Frisch]

1. Gives an interpretation satisfying the above constraints;
2. Gives an algorithm to decide the induced subtyping relation.
Polymorphic extension: adding type variables
Naive solution

\[ t ::= B | t \times t | t \rightarrow t | t \lor t | t \land t | \neg t | 0 | 1 \]
Naive solution

\[ t ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \]
Naive solution

\[
\begin{align*}
t & ::= B \mid t \times t \mid t \rightarrow t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha
\end{align*}
\]

Idea: Use the previous relation since is defined for “ground types”

Let \( \sigma : \text{Vars} \rightarrow \text{ClosedTypes} \) denote ground substitutions. Define:

\[
s \leq t \quad \overset{\text{def}}{\iff} \quad \forall \sigma . \, s\sigma \leq t\sigma
\]
Naive solution

\[ t ::= B \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

**Idea:** Use the previous relation since it is defined for “ground types”

Let \( \sigma : \text{Vars} \to \text{ClosedTypes} \) denote ground substitutions. Define:

\[ s \leq t \iff \forall \sigma. s\sigma \leq t\sigma \]

or equivalently

\[ s \leq t \iff \forall \sigma. [s\sigma] \subseteq [t\sigma] \]
Naive solution

\[ t ::= B \mid t \times t \mid t \to t \mid t \lor t \mid t \land t \mid \neg t \mid 0 \mid 1 \mid \alpha \]

**Idea:** Use the previous relation since it is defined for “ground types”

Let \( \sigma : Vars \to \text{ClosedTypes} \) denote ground substitutions. Define:

\[
\forall \sigma. s_\sigma \leq t_\sigma
\]

or equivalently

\[
\forall \sigma. [s_\sigma] \subseteq [t_\sigma]
\]

**THIS IS A WRONG WAY: TOO MANY PROBLEMS**
Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall \sigma. s\sigma \leq t\sigma$ is at least as hard as solving Diophantine equations.
Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall \sigma . s\sigma \leq t\sigma$ is at least as hard as solving Diophantine equations.

2. It breaks parametricity:
Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall \sigma. s\sigma \leq t\sigma$ is at least as hard as solving Diophantine equations.

2. It breaks parametricity:

$$ (t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t) $$

(1)
Haruo Hosoya conjectured that deciding $\forall \sigma . \ s\sigma \leq t\sigma$ is at least as hard as solving Diophantine equations.

It breaks parametricity:

$$\ (t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t) \ \ (1)$$

This inclusion holds if and only if $t$ is an indivisible type (eg., a singleton or a basic type):
Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall \sigma. s\sigma \leq t\sigma$ is at least as hard as solving Diophantine equations.

2. It breaks parametricity:

$$
(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)
$$

This inclusion holds if and only if $t$ is an indivisible type (e.g., a singleton or a basic type):

Property of indivisible types

If $t$ is an indivisible type, then for all possible interpretations of $\alpha$

$$
t \leq \alpha \quad \text{or} \quad \alpha \leq \neg t
$$

holds.
Problems with the naive solution

1. Haruo Hosoya conjectured that deciding $\forall \sigma. \, s \sigma \leq t \sigma$ is at least as hard as solving Diophantine equations.

2. It breaks parametricity:

$$ (t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t) \quad (1) $$

This inclusion holds if and only if $t$ is an indivisible type (eg., a singleton or a basic type):

- Property of indivisible types

  If $t$ is an indivisible type, then for all possible interpretations of $\alpha$

  $$ t \leq \alpha \quad \text{or} \quad \alpha \leq \neg t $$

  holds.

  - If $\alpha \leq \neg t$ then the left element of the union in (18) suffices;
  - If $t \leq \alpha$, then $\alpha = (\alpha \setminus t) \lor t$. Thus $(t \times \alpha) = (t \times (\alpha \setminus t)) \lor (t \times t)$. This union is contained component-wise in the one in (18).
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

holds if and only if \(t\) is *indivisible* is really catastrophic:
The fact that
\[(t \times \alpha) \leq (t \times \neg t) \vee (\alpha \times t)\]
holds if and only if \(t\) is *indivisible* is really catastrophic:
- Deciding subtyping needs deciding indivisibility ... which is very hard.
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

holds if and only if \(t\) is indivisible is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.
- This subtyping relation breaks parametricity: by subsumption a function generic in its first argument, becomes generic on its second argument.
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

holds if and only if \(t\) is \textit{indivisible} is really catastrophic:

- Deciding subtyping needs deciding indivisibility ... which is very hard.

- \textbf{This subtyping relation breaks parametricity}:
  by subsumption a function generic in its first argument, becomes generic on its second argument.

- A semantic solution was deemed unfeasible (even w/o arrows)
- Problem eschewed by resorting to syntactic solutions:
  [Hosoya, Frisch, Castagna: POPL 05], [Vouillon: POPL 06].
Problems with the naive solution

The fact that

\[(t \times \alpha) \leq (t \times \neg t) \lor (\alpha \times t)\]

holds if and only if \(t\) is indivisible is really catastrophic:

- Deciding subtyping needs deciding indivisibility … which is very hard.

- **This subtyping relation breaks parametricity:**
  by subsumption a function generic in its first argument, becomes generic on its second argument.

- A semantic solution was deemed unfeasible (even w/o arrows)
- Problem eschewed by resorting to syntactic solutions:
  [Hosoya, Frisch, Castagna: POPL 05], [Vouillon: POPL 06].

**A SEMANTIC SOLUTION IS POSSIBLE**
A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly.
A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly.

The crux of the problem is that for an indivisible type $i$

$$i \leq \alpha \quad \text{or} \quad \alpha \leq \neg i$$

validity can stutter from one formula to another, missing in this way the uniformity typical of parametricity.
A semantic solution

A faint intuition

The loss of parametricity is only due to the interpretation of indivisible types, all the rest works (more or less) smoothly.

The crux of the problem is that for an indivisible type $i$

$$i \leq \alpha \quad \text{or} \quad \alpha \leq \neg i$$

validity can stutter from one formula to another, missing in this way the uniformity typical of parametricity.

The leitmotif of this work

A semantic characterization of models where stuttering is absent, should yield a subtyping relation that is:

1. Semantic
2. Intuitive for the programmer
3. Decidable
A semantic solution

Rough idea

Make indivisible types “splittable” so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]
A semantic solution

Rough idea

Make indivisible types “splittable” so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

\[ \eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]
A semantic solution

Rough idea

Make indivisible types “splittable” so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

\[ \eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D}) \]

and now the interpretation function takes an extra parameter

\[ \llbracket \rrbracket : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})^{\text{Vars}} \rightarrow \mathcal{P}(\mathcal{D}) \]
A semantic solution

Rough idea

Make indivisible types “splittable” so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : \text{Vars} \rightarrow \mathcal{P}(\mathcal{D})$$

and now the interpretation function takes an extra parameter

$$[\ ] : \text{Types} \rightarrow \mathcal{P}(\mathcal{D})^{\text{Vars}} \rightarrow \mathcal{P}(\mathcal{D})$$

with

$$[\alpha]_\eta = \eta(\alpha)$$
$$[t_1 \lor t_2]_\eta = [t_1]_\eta \cup [t_2]_\eta$$
$$[0]_\eta = \emptyset$$

$$[\neg t]_\eta = \mathcal{D} \setminus [t]_\eta$$
$$[t_1 \land t_2]_\eta = [t_1]_\eta \cap [t_2]_\eta$$
$$[1]_\eta = \mathcal{D}$$
A semantic solution

**Rough idea**

Make indivisible types “splittable” so that type variables can range over strict subsets of every type, indivisible types included.

[intuition: interpret all non-empty types into infinite sets]

Since this cannot be done at syntactic level, move to the semantic one and replace ground substitutions by semantic assignments:

$$\eta : Vars \rightarrow \mathcal{P}(D)$$

and now the interpretation function takes an extra parameter

$$\llbracket \cdot \rrbracket : Types \rightarrow \mathcal{P}(D)^{Vars} \rightarrow \mathcal{P}(D)$$

with

- $$\llbracket \alpha \rrbracket_\eta = \eta(\alpha)$$
- $$\llbracket t_1 \lor t_2 \rrbracket_\eta = \llbracket t_1 \rrbracket_\eta \cup \llbracket t_2 \rrbracket_\eta$$
- $$\llbracket 0 \rrbracket_\eta = \emptyset$$
- $$\llbracket \neg t \rrbracket_\eta = D \setminus \llbracket t \rrbracket_\eta$$
- $$\llbracket t_1 \land t_2 \rrbracket_\eta = \llbracket t_1 \rrbracket_\eta \cap \llbracket t_2 \rrbracket_\eta$$
- $$\llbracket 1 \rrbracket_\eta = D$$

and such that it satisfies:

$$\llbracket t_1 \rightarrow s_1 \rrbracket_\eta \subseteq \llbracket t_2 \rightarrow s_2 \rrbracket_\eta \iff \mathcal{P}(\llbracket t_1 \rrbracket_\eta \times \llbracket s_1 \rrbracket_\eta) \subseteq \mathcal{P}(\llbracket t_2 \rrbracket_\eta \times \llbracket s_2 \rrbracket_\eta)$$
In this framework the natural definition of subtyping is

\[ s \leq t \overset{\text{def}}{\iff} \forall \eta. [s]_\eta \subseteq [t]_\eta \]

It “just” remains to find the uniformity condition to avoid stuttering and recover parametricity.
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \( \forall \eta. ([t \land \lnot \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \) — that is, \( (t \leq \alpha \text{ or } \alpha \leq \lnot t) \) — holds if and only if \( t \) is empty.
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

$$\forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset)$$

- **It avoids stuttering:** $\forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset)$ — that is, $(t \leq \alpha \text{ or } \alpha \leq \neg t)$ — holds if and only if $t$ is empty.

- **There are natural models:** all models that map all non-empty types into infinite sets satisfy it [*our initial intuition*].
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds:

\[ \forall \eta. (\llbracket t_1 \rrbracket \eta = \emptyset \text{ or } \llbracket t_2 \rrbracket \eta = \emptyset) \iff (\forall \eta. \llbracket t_1 \rrbracket \eta = \emptyset) \text{ or } (\forall \eta. \llbracket t_2 \rrbracket \eta = \emptyset) \]

- It avoids stuttering: \[ \forall \eta. (\llbracket t \land \neg \alpha \rrbracket \eta = \emptyset \text{ or } \llbracket t \land \alpha \rrbracket \eta = \emptyset) \] — that is, \( t \leq \alpha \text{ or } \alpha \leq \neg t \) — holds if and only if \( t \) is empty.

- There are natural models: all models that map all non-empty types into infinite sets satisfy it [our initial intuition].

- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.
The magic property: **convexity**

Consider only models of semantic subtyping in which the following convexity property holds

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \( \forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \) —that is, \( (t \leq \alpha \text{ or } \alpha \leq \neg t) \) — holds if and only if \( t \) is empty.

- There are natural models: all models that map all non-empty types into infinite sets satisfy it [our initial intuition].

- A sound, complete, and terminating decision algorithm: the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.

- An intuitive relation: the algorithm returns intuitive results (actually, it helps to better understand twisted examples)
The magic property: **convexity**

Consider **only** models of semantic subtyping in which the following **convexity** property holds

\[ \forall \eta. ([t_1] \eta = \emptyset \text{ or } [t_2] \eta = \emptyset) \iff (\forall \eta. [t_1] \eta = \emptyset) \text{ or } (\forall \eta. [t_2] \eta = \emptyset) \]

- It avoids stuttering: \( \forall \eta. ([t \land \neg \alpha] \eta = \emptyset \text{ or } [t \land \alpha] \eta = \emptyset) \) —that is, \( t \leq \alpha \text{ or } \alpha \leq \neg t \) — holds if and only if \( t \) is empty.

- **There are natural models:** all models that map all non-empty types into infinite sets satisfy it [our initial intuition].

- **A sound, complete, and terminating decision algorithm:** the condition gives us exactly the right conditions needed to reuse the subtyping algorithm devised for ground types.

- An intuitive relation: the algorithm returns intuitive results (actually, it helps to better understand twisted examples).
Examples of subtyping relations
We can internalize properties such as:

$$(\alpha \rightarrow \gamma) \land (\beta \rightarrow \gamma) \sim \alpha \lor \beta \rightarrow \gamma$$
Examples

We can internalize properties such as:

\[(\alpha \to \gamma) \land (\beta \to \gamma) \sim \alpha \lor \beta \to \gamma\]

or distributivity laws:

\[(\alpha \lor \beta \times \gamma) \sim (\alpha \times \gamma) \lor (\beta \times \gamma)\]
Examples

We can internalize properties such as:

\[(\alpha \to \gamma) \land (\beta \to \gamma) \sim \alpha \lor \beta \to \gamma\]

or distributivity laws:

\[(\alpha \lor \beta \times \gamma) \sim (\alpha \times \gamma) \lor (\beta \times \gamma)\]

and combining them deduce:

\[(\alpha \times \gamma \to \delta_1) \land (\beta \times \gamma \to \delta_2) \leq (\alpha \lor \beta \times \gamma) \to \delta_1 \lor \delta_2\]
Examples

We can internalize properties such as:

\[(\alpha \to \gamma) \land (\beta \to \gamma) \sim \alpha \lor \beta \to \gamma\]

or distributivity laws:

\[(\alpha \lor \beta \times \gamma) \sim (\alpha \times \gamma) \lor (\beta \times \gamma)\]

and combining them deduce:

\[(\alpha \times \gamma \to \delta_1) \land (\beta \times \gamma \to \delta_2) \leq (\alpha \lor \beta \times \gamma) \to \delta_1 \lor \delta_2\]

Of course the problematic relation never holds, whatever the \(t\):

\[(t \times \alpha) \not\leq (t \times \neg t) \lor (\alpha \times t)\]
We can prove relevant relations on infinite types, eg., for the type of generic $\alpha$-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \lor \text{nil}$$
We can prove relevant relations on infinite types, eg., for the type of generic $\alpha$-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \lor \text{nil}$$

we can prove that it contains both the $\alpha$-lists of even length

$$\mu z.(\alpha \times (\alpha \times z)) \lor \text{nil} \leq \mu z.(\alpha \times z) \lor \text{nil}$$

and the $\alpha$-lists with of odd length

$$\mu z.(\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}) \leq \mu z.(\alpha \times z) \lor \text{nil}$$
We can prove relevant relations on infinite types, e.g., for the type of generic $\alpha$-lists:

$$\alpha\text{-list} = \mu z. (\alpha \times z) \lor \text{nil}$$

we can prove that it contains both the $\alpha$-lists of even length

$$\mu z. (\alpha \times (\alpha \times z)) \lor \text{nil} \leq \mu z. (\alpha \times z) \lor \text{nil}$$

$\alpha$-lists of even length $\quad \alpha$-lists

and the $\alpha$-lists with of odd length

$$\mu z. (\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}) \leq \mu z. (\alpha \times z) \lor \text{nil}$$

$\alpha$-lists of odd length $\quad \alpha$-lists

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z. (\alpha \times (\alpha \times z)) \lor \text{nil}) \lor (\mu z. (\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}))$$
We can prove relevant relations on infinite types, eg., for the type of generic $\alpha$-lists:

$$\alpha\text{-list} = \mu z.(\alpha \times z) \lor \text{nil}$$

we can prove that it contains both the $\alpha$-lists of even length

$$\mu z.(\alpha \times (\alpha \times z)) \lor \text{nil} \leq \mu z.(\alpha \times z) \lor \text{nil}$$

and the $\alpha$-lists with of odd length

$$\mu z.(\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}) \leq \mu z.(\alpha \times z) \lor \text{nil}$$

and that it is itself contained in the union of the two, that is:

$$\alpha\text{-list} \sim (\mu z.(\alpha \times (\alpha \times z)) \lor \text{nil}) \lor (\mu z.(\alpha \times (\alpha \times z)) \lor (\alpha \times \text{nil}))$$

And we can prove far more complicated relations (see paper).
Subtyping algorithm
**Subtyping Algorithm:** \( t_1 \leq t_2 \)

**Step 1:** Transform the subtyping problem into an emptiness decision problem:

\[
t_1 \leq t_2 \iff \forall \eta. [t_1] \eta \subseteq [t_2] \eta \iff \forall \eta. [t_1 \land \neg t_2] \eta = \emptyset \iff t_1 \land \neg t_2 \leq \emptyset
\]
Subtyping Algorithm: $t_1 \leq t_2$

**Step 1:** Transform the subtyping problem into an emptiness decision problem:

$$t_1 \leq t_2 \iff \forall \eta. [t_1] \eta \subseteq [t_2] \eta \iff \forall \eta. [t_1 \land \neg t_2] \eta = \emptyset \iff t_1 \land \neg t_2 \leq 0$$

**Step 2:** Put the type whose emptiness is to be decided in disjunctive normal form.

$$\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}$$

where $a ::= b \mid t \times t \mid t \rightarrow t \mid 0 \mid 1 \mid \alpha$ and $\ell ::= a \mid \neg a$
Subtyping Algorithm: \( t_1 \leq t_2 \)

Step 1: Transform the subtyping problem into an emptiness decision problem:

\[
t_1 \leq t_2 \iff \forall \eta. [t_1] \eta \subseteq [t_2] \eta \iff \forall \eta. [t_1 \land \neg t_2] \eta = \emptyset \iff t_1 \land \neg t_2 \leq \emptyset
\]

Step 2: Put the type whose emptiness is to be decided in disjunctive normal form.

\[
\bigvee_{i \in I} \bigwedge_{j \in J} \ell_{ij}
\]

where \( a ::= b \mid t \times t \mid t \to t \mid 0 \mid 1 \mid \alpha \) and \( \ell ::= a \mid \neg a \)

Step 3: Simplify mixed intersections:

Consider each summand of the union: cases such as \( t_1 \times t_2 \land t_1 \to t_2 \) or \( t_1 \times t_2 \land \neg (t_1 \to t_2) \) are straightforward.

Solve:

\[
\bigwedge_{i \in I} a_i \bigwedge_{j \in J} \neg a_j' \bigwedge_{h \in H} \alpha_h \bigwedge_{k \in K} \neg \beta_k
\]

where all \( a \) are of the same kind.
Step 4: **Eliminate toplevel negative variables.**

\[ \forall \eta. \llbracket t \rrbracket \eta = \emptyset \iff \forall \eta. \llbracket t \{ -\alpha/\alpha \} \rrbracket \eta = \emptyset \]

so replace \( -\beta_k \) for \( \beta_k \) (for all \( k \in K \))

Solve:

\[ \bigwedge_{i \in I} a_i \bigwedge_{j \in J} \neg a'_j \bigwedge_{h \in H} \alpha_h \]
Step 4: Eliminate toplevel negative variables.

\[ \forall \eta. [t] \eta = \emptyset \iff \forall \eta. [t \{\neg \alpha / \alpha\}] \eta = \emptyset \]

so replace \( \neg \beta_k \) for \( \beta_k \) (forall \( k \in K \))

Solve:

\[ \bigwedge_{i \in I} a_i \bigwedge_{j \in J} \neg a'_j \bigwedge_{h \in H} \alpha_h \]

Step 5: Eliminate toplevel variables.

\[ \bigwedge_{t_1 \times t_2 \in P} \bigwedge_{h \in H} \alpha_h \leq \bigvee_{t'_1 \times t'_2 \in N} t'_1 \times t'_2 \]

holds if and only if

\[ \bigwedge_{t_1 \sigma \times t_2 \sigma} \bigwedge_{h \in H} \gamma^1_h \times \gamma^2_h \leq \bigvee_{t'_1 \sigma \times t'_2 \sigma} t'_1 \sigma \times t'_2 \sigma \]

where \( \sigma = \{(\gamma^1_h \times \gamma^2_h) \vee \alpha_h / \alpha_h\}_{h \in H} \)

(similarly for arrows)
Step 6: Eliminate toplevel constructors, memoize, and recurse.

Thanks to convexity and (set-theoretic) product decomposition rules

\[
\bigwedge_{t_1 \times t_2 \in P} t_1 \times t_2 \leq \bigvee_{t'_1 \times t'_2 \in N} t'_1 \times t'_2
\]

is equivalent to

\[
\forall N' \subseteq N. \left( \bigwedge_{t_1 \times t_2 \in P} t_1 \leq \bigvee_{t'_1 \times t'_2 \in N'} t'_1 \right) \text{ or } \left( \bigwedge_{t_1 \times t_2 \in P} t_2 \leq \bigvee_{t'_1 \times t'_2 \in N \setminus N'} t'_2 \right)
\]

(similarly for arrows)
Conclusion and New Directions
Conclusion

- We presented the first known solution to the problem of defining a semantic subtyping relation for a polymorphic regular tree types.

- A solution to this problem was considered unfeasible or even impossible.

- Our solution immediately applies to functional XML processing, but the potential fields of application seem much more numerous.

- Finally, our work opens both *practical* and *theoretical* new directions of research.
New typing possibilities:

fun even =
   | Int -> (x mod 2) == 0
   | _  -> x

Intuitively we want to type it by

\((\text{Int} \rightarrow \text{Bool}) \land (\alpha \backslash \text{Int} \rightarrow \alpha \backslash \text{Int})\)
Practical problems

New typing possibilities:

\[
\text{fun even } = \\
\quad | \text{Int } \rightarrow (x \text{ mod 2}) == 0 \\
\quad | \_ \rightarrow x
\]

Intuitively we want to type it by

\[
(\text{Int} \rightarrow \text{Bool}) \land (\alpha \text{\ Int } \rightarrow \alpha \text{\ Int})
\]

Local type inference:

Let \texttt{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list } \rightarrow \beta \text{ list},

then for \texttt{map even} we wish to deduce the following type:

\[
(\text{Int list } \rightarrow \text{Bool list}) \land \\
((\alpha \text{\ Int}) \text{ list } \rightarrow ((\alpha \text{\ Int}) \text{ list }) \land \\
(\alpha \text{ list } \rightarrow (((\alpha \text{\ Int}) \lor \text{Bool}) \text{ list })
\]
Practical problems

**New typing possibilities:**

```ml
fun even = |
| Int -> (x mod 2) == 0
| _   -> x
```

Intuitively we want to type it by

\[(\text{Int} \rightarrow \text{Bool}) \land (\alpha \downarrow \text{Int} \rightarrow \alpha \downarrow \text{Int})\]

**Local type inference:**

Let \(\text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}\),

then for \(\text{map even}\) we wish to deduce the following type:

\[(\text{Int list} \rightarrow \text{Bool list}) \land (\alpha \downarrow \text{Int list} \rightarrow (\alpha \downarrow \text{Int} \downarrow \text{list}) \land (\alpha \text{ list} \rightarrow ((\alpha \downarrow \text{Int}) \lor \text{Bool}) \text{ list})\)

\(\text{int lists return bool lists}\)
Practical problems

New typing possibilities:

fun even =
    | Int  -> (x mod 2) == 0
    | _    -> x

Intuitively we want to type it by

\((\text{Int} \rightarrow \text{Bool}) \land (\alpha \setminus \text{Int} \rightarrow \alpha \setminus \text{Int})\)

Local type inference:

Let \text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list},
then for \text{map even} we wish to deduce the following type:

\((\text{Int list} \rightarrow \text{Bool list}) \land
(\alpha \setminus \text{Int}) \text{ list} \rightarrow (\alpha \setminus \text{Int}) \text{ list}) \land
(\alpha \text{ list} \rightarrow ((\alpha \setminus \text{Int}) \lor \text{Bool}) \text{ list})\)
Practical problems

New typing possibilities:

fun even =
  | Int -> (x mod 2) == 0
  | _   -> x

Intuitively we want to type it by

\((\text{Int} \rightarrow \text{Bool}) \land (\alpha \cdot \text{Int} \rightarrow \alpha \cdot \text{Int})\)

Local type inference:

Let \(\text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}\),

then for \(\text{map even}\) we wish to deduce the following type:

\((\text{Int list} \rightarrow \text{Bool list}) \land (\alpha \cdot \text{Int list} \rightarrow (\alpha \cdot \text{Int list}) \land (\alpha \cdot \text{list} \rightarrow ((\alpha \cdot \text{Int list} \lor \text{Bool list}) \land \text{int lists return bool lists} \land \text{lists w/o ints return the same type} \land \text{ints in the argument are replaced by bools})\)
Practical problems

New typing possibilities:

```haskell
fun even =
  | Int -> (x mod 2) == 0
  | _   -> x
```

Intuitively we want to type it by

\[(\text{Int} \rightarrow \text{Bool}) \land (\alpha \downarrow \text{Int} \rightarrow \alpha \downarrow \text{Int})\]

Local type inference:

Let \(\text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}\),

then for \(\text{map even}\) we wish to deduce the following type:

\[
(\text{Int list} \rightarrow \text{Bool list}) \land
((\alpha \downarrow \text{Int}) \text{ list} \rightarrow (\alpha \downarrow \text{Int}) \text{ list}) \land
(\alpha \text{ list} \rightarrow ((\alpha \downarrow \text{Int}) \lor \text{Bool}) \text{ list})
\]

int lists return bool lists
lists w/o ints return the same type
ints in the argument are replaced by bools

Cannot be obtained by just instantiating the type of \(\text{map}\)
Practical problems

New typing possibilities:

fun even =
    | Int -> (x mod 2) == 0
    | _   -> x

Intuitively we want to type it by

\((\text{Int} \to \text{Bool}) \land (\alpha \setminus \text{Int} \to \alpha \setminus \text{Int})\)

Local type inference:

Let \text{map} : (\alpha \to \beta) \to \alpha\ \text{list} \to \beta\ \text{list},

then for \text{map even} we wish to deduce the following type:

\(\text{(Int list} \to \text{Bool list}) \land\)
\((\alpha \setminus \text{Int})\ \text{list} \to (\alpha \setminus \text{Int})\ \text{list}) \land\)
\(\alpha\ \text{list} \to ((\alpha \setminus \text{Int}) \lor \text{Bool})\ \text{list})\)

\(\text{int lists return bool lists}\)
\(\text{lists w/o ints return the same type}\)
\(\text{ints in the argument are replaced by bools}\)

Cannot be obtained by just instantiating the type of \text{map}

No principal typing (needs infinite connectives)
Practical problems

New typing possibilities:

```
fun even =
| Int -> (x mod 2) == 0
| _  -> x
```

Intuitively we want to type it by

\[(\text{Int} \rightarrow \text{Bool}) \land (\alpha \rightarrow \alpha)\]

Local type inference:

Let \(\text{map} : (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}\),
then for \(\text{map even}\) we wish to deduce the following type:

\[(\text{Int list} \rightarrow \text{Bool list}) \land
(\\alpha\text{\ Int list} \rightarrow \\alpha\text{\ Int list}) \land
(\alpha \text{ list} \rightarrow ((\alpha \rightarrow \text{Bool list}))\text{\ Int list})\]

Cannot be obtained by just instantiating the type of \(\text{map}\)

No principal typing (needs infinite connectives)
Convexity and parametricity?

In reality, the condition to be used is the generalization to $n$ types:

$$\forall \eta. ([t_1]_\eta = \emptyset \text{ or } \cdots \text{ or } [t_n]_\eta = \emptyset) \iff (\forall \eta. [t_1]_\eta = \emptyset \text{ or } \cdots \text{ or } (\forall \eta. [t_n]_\eta = \emptyset))$$
Convexity and parametricity?

In reality, the condition to be used is the generalization to $n$ types:

$$\forall \eta.([t_1]_{\eta}=\emptyset \text{ or } \cdots \text{ or } [t_n]_{\eta}=\emptyset)$$

$$\iff$$

$$\left(\forall \eta. [t_1]_{\eta}=\emptyset\right) \text{ or } \cdots \text{ or } \left(\forall \eta. [t_n]_{\eta}=\emptyset\right)$$

The big question

What is the relation of the condition above with parametricity?
Is it a language-independent semantic characterization of it?
Convexity and parametricity?

In reality, the condition to be used is the generalization to $n$ types:

$$\forall \eta.([t_1] \eta = \emptyset \text{ or } \cdots \text{ or } [t_n] \eta = \emptyset)$$

$$\iff$$

$$(\forall \eta. [t_1] \eta = \emptyset) \text{ or } \cdots \text{ or } (\forall \eta. [t_n] \eta = \emptyset)$$

The big question

What is the relation of the condition above with parametricity?
Is it a language-independent semantic characterization of it?

Two examples of uniformity:

- $(t_1 \times \cdots \times t_n)$ is empty if and only if exists at least one $t_i$ empty
- Definability in the second-order typed $\lambda$-calculus harnesses expressions to behave uniformity. Similarly, convexity semantically harnesses the denotations of expressions and forces them to behave uniformly.
Convexity and parametricity?

In reality, the condition to be used is the generalization to \( n \) types:

\[
\forall \eta. ([t_1] \eta = \emptyset \text{ or } \ldots \text{ or } [t_n] \eta = \emptyset) \iff \\
(\forall \eta. [t_1] \eta = \emptyset) \text{ or } \ldots \text{ or } (\forall \eta. [t_n] \eta = \emptyset)
\]

The big question

What is the relation of the condition above with parametricity? Is it a language-independent semantic characterization of it?

Two examples of uniformity:

- \((t_1 \times \ldots \times t_n)\) is empty if and only if exists at least one \( t_i \) empty
- Definability in the second-order typed \( \lambda \)-calculus harnesses expressions to behave uniformly. Similarly, convexity semantically harnesses the denotations of expressions and forces them to behave uniformly.

... we have strong flavors of parametricity